AFRL-IF-RS-TR-2004-261
Final Technical Report
September 2004

IMPROVING COALITION PERFORMANCE BY EXPLOITING PHASE TRANSITION BEHAVIOR

Computational Intelligence Research Laboratory

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. K273

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2004-261 has been reviewed and is approved for publication

APPROVED: /s/

EDWARD DEPALMA
Project Engineer

FOR THE DIRECTOR: /s/

JAMES W. CUSACK, Chief
Information Systems Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	SEPTEMBER 2004	Final Jun 00 – Nov 03

4. TITLE AND SUBTITLE
IMPROVING COALITION PERFORMANCE BY EXPLOITING PHASE TRANSITION BEHAVIOR

5. FUNDING NUMBERS
C - F30602-00-2-0534
PE - 62702F
PR - ANTS
TA - 00
WU - 03

6. AUTHOR(S)
David W. Etherington and
Andrew J. Parkes

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Oregon
CIRL, 1269 University of Oregon
Eugene Oregon 97403-1269

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington Virginia 22203-1714

AFRL/IFSA
26 Electronic Parkway
Rome New York 13441-4514

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2004-261

11. SUPPLEMENTARY NOTES
AFRL Project Engineer: Edward DePalma/IFSA/(315) 330-7454/ Edward.DePalma@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***
This document describes research into the effects of phase transitions and related phenomena on the design and operation of autonomous negotiating teams (ANTs). Results are reported in three areas: computational thresholds, solution clusters, and pseudo-Boolean solvers. First, the existence of computational thresholds below the usual computational hardness phase transition is shown, and the implications of these thresholds for "anytime", "good enough soon enough" behavior are discussed. Second, the effects of solution-clustering behavior on problem decomposition and negotiation strategy are explored, and it is shown that focusing negotiations on clusters both increases the chance of success and reduces the number of rounds of negotiation required.
Finally, the application of pseudo-Boolean representations and solvers to ANTS problems is discussed, including the impacts of various heuristic choices and learning strategies that were necessary to provide practical leverage on the ANTs program's demonstration problems.

14. SUBJECT TERMS
Phase Transitions, Solution Clusters, Negotiation, Computational Thresholds, Resource Management, Agent Tasking, Search, Robustness, Constraint Satisfaction

15. NUMBER OF PAGES
30

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

TABLE OF CONTENTS

LIST OF FIGURES

1. Overview

Solving real military problems often requires solving hard optimization problems, such as scheduling, under stringent bounds on computation time. This usually requires us to abandon the hope of optimal solutions. Instead 'anytime solvers" are used: solvers that can rapidly produce some solution, possibly of low quality, then go on to produce better quality solutions if given more time. The common experience with anytime solvers is that the solution improves rapidly at first, but eventually progress slows down drastically. As the solution quality approaches optimality the computational effort required to make further improvements increases dramatically. This behavior is actually inherent in the problems themselves. When progress begins to require vastly increased resources we refer to this as a computational "cliff".

In a monolithic solver (one agent working on one CPU) avoiding such cliffs requires changing the solution algorithms themselves. In distributed, or multi-agent, systems the greater freedom of choice in building the system gives us new opportunities to avoid the cliffs.

The study of "phase transitions" is relevant because they concern cases in which small changes in a parameter, such as temperature, cause large changes in the resulting system, for example changing water into ice. These sharp changes correspond to the sharp changes in properties of optimization systems near optimality: a small increase in the quality demanded of a solution can require a large change in computational effort, or if we are already optimal it can push us from a solvable problem to an unsolvable problem.

This effort sought to address the impacts of this type of behavior on systems comprised of Autonomous Negotiating Teams (ANTs). We were able to show new results on the relevance of phase transitions and other algorithmic threshold phenomena to coalition performance and negotiation strategies. We also provided computational tools to help both theoretical development and practical demonstrations of ANTs technology.

Chapter 2 discusses results that were obtained with respect to phase transitions and thresholds and their relevance in guiding ANTs systems. In particular, we showed that marked transitions from easy to difficult computational behavior occur earlier than expected, and that these transitions have important implications for ANTs behavior.

Chapter 3 discusses applications of clustering behavior exhibited by solution spaces, and their implications to the negotiation structure required for efficient coalition behavior in ANTs systems. The objective of this aspect of the work was to take a practical, incremental, approach to guidance of ANT coalition formation and problem decomposition that draws on already available (or easily obtainable) information to avoid both unfortunate problem decompositions and inappropriate agent assignments that would otherwise lead to unacceptable computational behavior. This work provides the beginnings of a formal understanding of the interaction between decomposition and phase transition phenomena in distributed problems, using various problem decomposition methods.

Chapter 4 focuses on a particular solver architecture, the PARIS solver, and its underlying pseudo-Boolean representation language, as well as an optimizing version, oPARIS. PARIS and oPARIS both provide tools for experimental exploration of the problem space and transition behavior, and provide computational support for the practical demonstrations associated with the overall ANTs program.

2. Phase Transitions and Algorithmic Thresholds

2.1 Introduction

Everyone is familiar with the way that the properties of physical systems can change abruptly as we vary their temperature. The properties of H_2O at -1°C (ice) is quite different from that a +1°C (liquid water), whereas, a 2°C change makes little difference at other temperatures until we reach 99°C. Thus as we vary the control parameter, temperature, the properties exhibit phases: e.g., ice, water, steam. Inside a phase, the properties of the systems change slowly, but there are abrupt changes changes between phases.

The abruptness of such changes is due to the large number of degrees of freedom within the system—e.g., the large number of water molecules. Systems of small numbers of molecules do not show phase transitions. That is, the statistical properties of very large systems can show effects that are not apparent in small systems. We are familiar with this for physical systems, but it can also happen for non-physical systems, and in particular for systems of interacting constraints. Hence, such systems are relevant to decision and optimization problems.

An example that we often use in studies is that of a phase transition in satisfiability problems, and specifically that of Random 3SAT, a problem that has been extensively studied. In this case the control parameter, t, is the ratio of clauses to variables, and the property, P, that we care about is whether or not the instance has a satisfying assignment. Given an algorithm to determine satisfiability, the typical behavior is shown in Figure 2.1, with a phase transition at $t_C \approx 4.2$.

The important aspect here is that the phase transition associated with NP-hard properties, such as satisfiability, typically splits up the parameter space into 3 regions:

- **easy:** $t < t_C$. Instances are almost always satisfiable, that is, the chance that they are unsatisfiable is "infinitesimal"[1]. Furthermore, it is relatively easy to find a solution.
- **hard:** $t \approx t_C$ these critical instances have finite probability of being satisfiable (sat), and the same for unsatisfiable (unsat). This region is the hardest to solve, because instances, and the sub-problems that arise during search, are always "on the edge between sat and unsat" and distinguishing on which side they lie is hard.
- **less-hard:** $t > t_C$ (often inaccurately called "easy" again). Here problems are almost always unsatisfiable. Solving them is much easier than for the critical region, but still (typically) significantly harder than for the "easy" region.

Significantly, the peak in the search cost seems to apply to many (possibly all) algorithms. It is algorithm independent and is associated with a transition in the properties of the instances.

[1] Of course, this all has a precise statement, which can be found in the literature

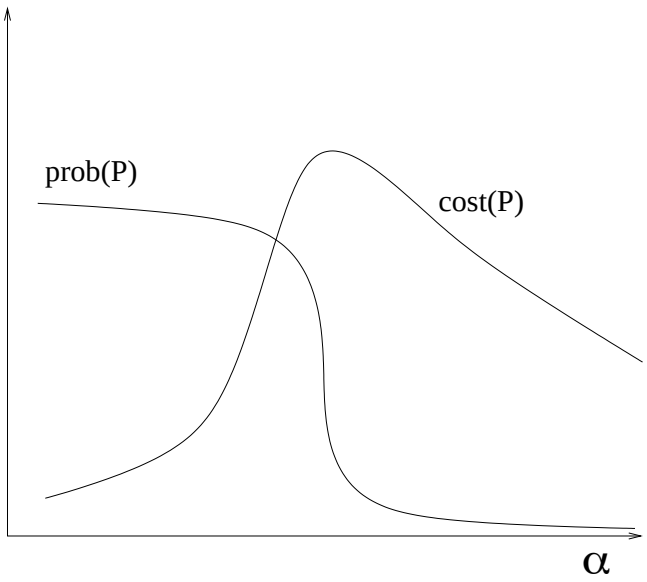

Figure 2.1: Typical behavior at a phase transition of some property "P" such as "satisfiable." The x-axis is the value of some control parameter, t, for example, the clause/variable ratio. The line marked "prob(P)" is the probability that P holds in some randomly selected instance at t. The line "cost(P)" is the average search cost of determining whether or not P holds for an instance.

Some of our work has been directed at refining this picture from the point of view of picking out and clarifying aspects that are relevant to ANTS, that is to search that might be distributed, but that tends to focus on obtaining fast solutions in the "easy" region.

Our contributions are discussed in three papers, which are summarized in the next three sections. The first paper [20] looks at whether the easy region is as easy as one might hope it should be for distributed algorithms. The second paper [22] attempts to quantify the meaning of "easy." Finally, the third paper [23] follows up and shows a surprising change in scaling behavior that occurs within the easy region—away from the phase transition. We will finish this high-level review of CIRL work with a discussion of the import of these results for ANTs systems.

2.2 Complexity of Distributed Local Search

This paper [20] asks the question:

> As we move away from the phase transition region into the underconstrained region, do the problems become efficiently solvable, on average, by parallel (as well as sequential) algorithms?

For a sequential algorithm, "efficient" usually means polynomial in the number of variables ($poly(n)$). However, for a parallel algorithm, we want to take good advantage of parallel computation, and so "efficient" is generally taken to mean being polynomial in the log of the number of variables ($poly(\log(n))$) or "polylog time".

The question is non-trivial because there are problems (called P-complete) that can be solved efficiently sequentially, but are generally believed to have no efficient solution. It could have

happened that the easy problems below the phase transition had no efficient parallel solution, and this would have bad implications for the underpinnings of ANTs. However, in fact we found (empirically) that they did have efficient parallel solutions.

In particular, we studied the behavior of a distributed local search, algorithm, WSAT, since such algorithms are fairly commonly used to solve distributed optimization problems. On Random 3SAT, WSAT exhibited polylog scaling, specifically we found that distributed WSAT is O($\log(n^2)$).

2.3 Scaling of WalkSAT

In [22] we study the scaling over the entire easy region. In comparison most previous studies have been restricted to just the phase transition region. Over most of this region the scaling of the (sequential) local search algorithm WalkSAT is found to be a simple power-law (equation 12 of [22]). Specifically, the number of flips needed scales as:

$$\frac{g_0 n}{(\alpha_0 - \alpha)^{g_1}} \tag{2.1}$$

where n is the number of variables, α is the clause/variable ratio, and g_0,g_1 and α_0 are constants.

The utility of this simple scaling law is to suggest that the scaling behavior of even difficult-to-predict algorithms can empirically have good behaviors.

We expect this line of work to be developed in the following directions:

- scaling laws might prompt some theoretical work—it seems reasonable that behavior ought to be predictable in the easy region
- provide an example of an empirical law that might be useful in other domains, and so might be useful for empirical predictions of scaling behavior.

Note that this formula is valid only for $\alpha < \alpha_0$, but that for any such α that the scaling is linear.

However, the surprise was that experiments yielded that $\alpha_0 \approx 4.19$ was significantly below the phase transition point. This suggested that there could be a significant change in the scaling behavior at a point distinctly below the phase transition.

2.4 Scaling of Pure WalkSAT

Unfortunately, although [22] suggested the existence of a threshold below the phase transition, the algorithm used was so effective that the potential threshold was so close to the phase transition that separating it from the phase transition was experimentally difficult. Hence, in [23] we studied an algorithm that was heuristically less effective, and so likely to have a threshold further from the phase transition. We also selected for study an algorithm that is very simple and so might (we hope) ultimately be amenable to theoretical analysis.

The algorithm used is that of "Pure Pure Random Walk" (PRWSAT):

$P :=$ a randomly generated truth assignment
for $j := 1$ to MAX-FLIPS $\{$
  if (P is a solution) **then** return P
  else

$c :=$ a randomly selected unsatisfied clause
flip a random literal from c }
return failure

This algorithm [17] predates (and inspired) the usual WalkSAT. Somewhat remarkably, despite its naivete, PRWSAT is average polytime on 2SAT (which has a linear time algorithm, but it is still somewhat surprising that such a naive local search algorithm does not end up making exponentially long mistakes).

We find experimental indications of a threshold at $\alpha \approx 2.7$:

- $\alpha < 2.7$ The scaling of PRWSAT is simple linear. Furthermore, the variance of runtimes between different instances is small. That is, in this region, the runtime of the local search algorithm is not only fast, but is also highly predictable between instances.
- $\alpha > 2.7$. In this region the scaling appears to be bad (not linear, and possibly not even polynomial). Furthermore, the variance of runtime between instances is no longer small. That is, the scaling is bad, and there is also little ability to predict the runtime on an instance.

Note that the threshold is different than for the heuristically-enable algorithm studied earlier. That is, the position of the threshold is algorithm dependent, in contrast to the algorithm independence of the phase transition itself.

This shows the existence of an algorithm-dependent threshold, that separates regions into:

- **good** linear or polytime scaling, low variance between instances. Runtimes are low and predictable
- **bad** bad (exponential) scaling, high variance between instances. Runtimes are high and unpredictable.

Clearly, for ANTS systems, in which we want to be able to predict runtimes, we want to be below the threshold for the algorithm.

2.5 Conclusions

Our belief before this work—a reflection of the general belief of the community—was that, while the algorithms slow down as we approach the phase transition, they would do so gradually. Hence, the main lesson learned was that sharp transitions, from good scaling to bad scaling properties can also occur when using a local search method. Furthermore, the location of the threshold can vary with the details of the algorithm—in contrast to the phase transition itself which is algorithm independent.

Better algorithms presumably have a threshold that is closer to the phase transition, but will require more computational effort. Hence there is a tradeoff:

- high quality algorithm—this is slower, but might still be a net win because the problem ends up in the polynomial scaling region.
- low quality algorithm—faster: if the solution quality expected to be achievable in the time limit is low enough; then we might effectively be in the easy region, and so the resulting polynomial scaling means the algorithm will produce good enough results soon enough. This too might be a net win, because of the reduced communication costs in implementing the simpler algorithm.

A reasonable goal would be to estimate the solution quality likely to be achievable in the time limit, and then use this to select the cheapest algorithm that would be in its easy region at that quality.

3. Exploiting Solution-Space Structure

3.1 Introduction

Suppose we have some combinatorial optimization or decision problem and an associated, exponentially large, search space. Typical examples include propositional satisfiability and scheduling. Such problems are often hard because the number of solutions is typically much less than the size of the search space. That is, solutions are very rare within the search space. However, earlier studies of problem instances from the phase transition region of random satisfiability problems [18] showed that, despite their rareness, the solutions still tend to cluster together. Thus, by a "solution cluster" we are simply referring to a cluster of solutions within the overall search space: that is, a region of the search space that is rich in solutions, despite their overall scarcity.

It seems to be natural that many problems will have solution clusters. A familiar example of this would be in standard scheduling problems. Suppose we demand that the schedule be optimal, in the sense of having the shortest possible overall length. Optimal schedules are very rare within the space of all schedules. However, despite this rarity, the optimal schedule is likely to be part of a solution cluster. To see this, consider the critical path within the schedule; these are the tasks whose start times cannot be perturbed without increasing the overall schedule length. Tasks that are not on the critical path can, by definition, be moved to some extent without breaking optimality. If the optimal solution were a single isolated solution then all tasks would be on the critical path. In practice, this is very unlikely. An optimal solution is likely to be surrounded by a solution cluster corresponding to the results of moving tasks not on the critical path.

In the following we will explain why this is relevant to negotiating systems. In all cases we will take a high-level descriptive and motivational approach, and refer to our paper [21] for formal results, and further details. Unlike that paper, however, in this context we will be discussing the communication of solution clusters. It is essential that this does not lead to excessive communication costs; and so the set of solutions should have a compact representation. Hence, we adopt the following definition.

Definition 3.1.1 *A* solution cluster *is a region of the search space that:*

- *contains a large (exponential) number of solutions*
- *can be represented compactly; that is, has a polynomial-size representation. Ideally, the representation should not be significantly larger than that for a single solution.*

We emphasize that the "definition" is not meant formally, but merely in a descriptive sense. In practice, the exact form of the definition will need to be tailored to the domain and system requirements.

3.2 Solution Clusters, Negotiations, and Avoiding Distractions

Suppose that we have a "system-of-systems". For simplicity we will consider just two communicating systems, A and B. A good example to keep in mind would be the SNAP system [16] interacting with the MAPLANT system [24, 25].

Suppose that systems (or agents) A and B are cooperating to try to find a good solution to a hard optimization/decision problem. Both A and B have access to their portion of the overall problem, but neither can see the entire problem. Since the portions interact, the systems cannot find solutions alone, but must ensure that their solutions are compatible. Typically, the search within a single system will involve constructing solutions: working from partial assignments (of values to variables) and trying to build a complete solution. Optionally, it might also involve attempts to find solution clusters (we will describe how to do this later). At any stage, an agent could communicate its partial results to the other agent(s). However, the act of communication is expensive. Besides the transmission costs, there is also the possibility of distracting the other agent from its own work.

The general issue here is that of deciding how much work to do before communicating your (partial) results to others. In human terms, at one end of the spectrum, one might communicate even the slightest progress to others, but this might only result in distracting them from work they need to be doing, with little positive effect. At the other end of the spectrum, one could work privately until having what seems to be a perfect solution, based on the data you know, only to discover that another agent has established some constraint that clearly invalidates the entire proposed solution.

In terms of search, the issue is which one of the following to use for communications:

1. Partial solutions. In the extreme case, every time the partial assignment is extended and does not result in internal failure (that is, it seems fine internally), then it could be transmitted as a potential portion of a solution.
2. Single solutions. Once the system has a complete satisfying assignment then it is transmitted.
3. "Solution clusters". Even after finding a single solution, the system could continue to work and convert it into a solution cluster before transmitting.

(In reality, these are just the natural points within a whole spectrum of choices; for example, one could decide to just transmit whatever solution one had at some regular time interval.)

All of these have their pros and cons, as follows:

1. **Partial solutions.**

   Pros Early Pruning. The advantage of early communication is that bad regions of the search space can be pruned early. System A might spend of a lot of effort on a partial assignment P, even though B might know that P is fatally flawed and should be immediately discarded. However, part of the system design should be to partition the overall problem in such a way as to reduce the chances of this happening.

   Cons Distraction of other agents. If the problem is well-partitioned then most partial assignments will not be pruned by the other agents. In this case, the frequent transmission of partial assignments will merely distract the other agent from its own search problem, and not improve the search. In an extreme case the changes inside one agent might correspond to fixing values within some very easy sub-problem, and so be essentially irrelevant to the other agent(s).

2. **Single solutions.** This is the standard choice.

Pros Once we have a local solution, it is natural and simple to try to extend it to a global solution by sending it to other agents.

Cons It is possible that the single solution from A will be incompatible with B's solution due to many "trivial" reasons. The resulting negotiations to fix these "merging defects" might be straightforward, but they still might correspond to many rounds of negotiations, with associated increase in communications costs.

3. **Solution Clusters.**

Pros The intended advantage of sending a solution cluster is that it is an inexpensive way to send an exponential number of solutions to the other agent. The other agent can then select one that is most compatible with its own solution. Since it has, effectively, an exponential number of choices, the hope is that the chance of defects is much reduced. The paper [21] provides some experimental support for this hoped-for advantage.

Cons After all the effort to find a solution cluster, another agent might reply that there is a simple defect that invalidates the entire cluster, and so corresponds to the loss of all the associated computational effort. For this reason, it is important that the computational cost of finding a cluster not be too high.

Note that the notion of distraction is well-known in the community, (see, for example, [14]) but has been limited to cases of partial assignments and single solutions. The work at CIRL allowed the spectrum of communication choices to be increased, and allows even lower distraction by permitting communications to be delayed until a solution cluster is available.

3.3 Solution Clusters in SAT

Given the potential advantages of solution clusters, the immediate questions are:

- how can one more precisely define clusters?
- how can one find clusters?
- are their advantages actually realized?

The paper [21] answers these questions for the case where each system is a satisfiability problem. Brief summaries of those results follow.

3.3.1 Defining solution clusters for SAT

In SAT we define a solution cluster in terms of a partial assignment P (that is, a set of values for a subset of all the variables). The residual theory from P is just the set of constraints that are not satisfied by P, that is, the constraints that remain between the variables not assigned a value by P.

For P to correspond to a solution cluster, we simply require that the residual theory is underconstrained. That is, it should not only be satisfiable, but (informally) have many solutions, and they should be very easy to find. In SAT, this corresponds to the residual theory having a low clause to variable ratio. Optionally, we might also require that the residual theory not force any residual variable to a single value, otherwise this forced value should simply be added to P.

The actual definition is complicated somewhat be the fact that not all variables in a system might be visible to other systems. Variables can be split into private and public, and the relevant communication of a solution cluster should refer only to the public variables.

3.3.2 Finding Solution Clusters

The simplest algorithm is a bottom up greedy search. It starts with a single solution found using some standard search procedure, and then iteratively tries to "relax" to a solution cluster by unsetting variables, trying to select variables that allow the largest cluster, and stopping when the residual theory is about to become too constrained:

Procedure 3.3.1 (Find-Solution-Cluster) *Given a* SAT *problem* C*, to compute a partial assignment solution cluster:*

1 Find a single solution, $P \leftarrow \text{SOLVE}(C)$
  else return failure
2 **while** ($resid(P)$ is under-constrained)
3 P' = P
4 select variable $x \in P$ such that resid(P-x) is least constrained
5 P= P-x
6 **return** P' (the latest under-constrained P)

Detecting the constrainedness of resid(P-x) is usually just a matter of counting unsatisfied clauses, and hence is relatively fast. If necessary, the variables x that are tested can be restricted to come from some small, heuristically selected set, rendering the algorithm even faster. There is also no attempt here to find a maximum sized cluster, the greedy search simply finds some cluster that is locally maximally sized. It is important to note that the proposed usage of solution clusters does not in any way require that they be optimal.

Again this was modified somewhat by the need to consider the public and private variables, but the essential idea, a bottom-up greedy relaxation, still applies.

3.3.3 Experimental Results on Solution Clusters

We considered a simple scenario of multiple systems trying, in a single round of negotiations, to find a global solution. The success metric was simply the probability of success. Results showed that, in the interesting cases in which the problem was neither too easy nor insoluble, that passing solution clusters drastically reduced the probability of failure of the negotiations.

3.4 Relevance to SNAP–MAPLANT

The existing work [21] only considered the case in which the constraints are represented as a satisfiability problem. In the real world, constraints are likely to have a much more complicated representation, however, we expect that the essential idea "find a single solution and then greedily relax it to solution cluster" will still be applicable.

In the case of MAPLANT, the constraints are solved by use of the constraint-programming (CP) system, Mozart, hence here we sketch out how such an approach might be used in a CP system. In implementing the algorithm in CP we can of course use the existing search within CP in order to find the initial system. In CP, there is usually no direct support for unsetting a variable, however, this is presumably straightforward to implement by building up a separate computation based on the partial assignment, $P - x$, to be tested.

In CP, the most difficult task might be to implement a measure of the constrainedness of the residual theory. Presumably, it is possible to detect which constraints are already satisfied by P. It is tempting to then just keep a count of the unsatisfied constraints. If all the constraints are primitive (built-in, simple, constraints) then this is reasonable. However, in CP, one makes extensive use of global constraints (such as the "cumulative constraint" that is typically used to enforce resource bounds in scheduling problems). These involve many, if not all, of the variables of the problem, and it is unlikely that they will be totally satisfied. It is likely that methods will need to be developed to determine whether the residual global constraints are loosely constrained or not. However, such information is likely to also be of use in building heuristics for the initial search; for example, it is common to try to select the least constraining values, and so measurements of "constrainedness" are likely to be of general use.

Also, in CP, it is usual to use a finite-domain encoding rather than a boolean encoding. In this case a natural definition of a solution cluster would be to view the partial assignment as a set of unary constraints on variables, and instead of only allowing equality constraints, to also allow inequalities. For example, P might be $\{x \leq 3, y = 8, y \geq 2\}$. The consequences of such unary constraints will be determined by the inbuilt propagation methods in CP systems, and the resulting ranges on variables will correspond to the residual theory.

3.5 Conclusions

This chapter provided an overview of the theory of solution clusters, their relevance to negotiation, methods to find them, and potential applicability to systems of systems such as SNAP-MAPLANT.

4. Expressiveness and Utility of Generic Representation Schemes

4.1 Introduction

A standard approach to solving decision/optimization problems is to express them in a general purpose language for which specialized solvers are available. In operations research (OR), integer programming (IP) formulations are commonly used: variables are integer-valued, and constraints are simple linear inequalities over these variables. In artificial intelligence (AI), it has been more common to use the logical formalism of propositional satisfiability (SAT), in which variables are Boolean (true or false) and constraints are clauses—disjunction of literals, where a literal is a variable or its negation: for example, $\neg x \vee y \vee \neg z$.

Both formulations, IP and SAT, are NP-complete and so are suitable for encoding problems in NP. Furthermore there are many well-developed solvers for both. Generally speaking IP seems to be best for problems with a significant arithmetic component—meaning that someone solving them might do a lot of counting and arithmetic reasoning. SAT seems better for cases where logical reasoning is more natural: "if x then either y or z is true, and hence ...". However, there are many problems that seem to be a mix of both arithmetic and logical reasoning. Such problems present two difficulties:

- **IP solvers are bad at logic.** SAT solvers are good at logical problems because they do a lot of logical inference internally, and find logical consequences of the constraints that can be used to prune out unnecessary search. Some OR solvers have some aspects of inference (those doing branch-and-cut) but generally inference is not emphasized.
- **SAT solvers are bad at arithmetic.** In fact, it is inherent in the nature of the representation that SAT solvers are spectacularly bad at arithmetic. The best example is that of the propositional pigeonhole principle (PHP); the PHP simply says that it is not possible to put $n + 1$ pigeons into n holes without any pigeons sharing a hole. This is obvious by simple arithmetic, but, no matter how clever their heuristics, SAT solvers provably take exponential time to prove the PHP unsolvable.

Our goal has been to develop a solver that is good at both arithmetic and logic and so is suitable for some "intermediate problems". For example, some arithmetic/logic problems of interest to ANTS arise in SNAP [16]. The arithmetic aspects arise from the resource bounds—counting naturally occurs to see whether the tasks can fit within the resource limits. The logical aspects arise from dependencies between tasks—for example, pilots need to perform some missions in order to obtain qualifications permitting them to do other missions.

To enable these investigations, we selected the representational language of "pseudo-Boolean" (PB) constraints, which is a simple intermediate between IP and SAT. PB has:

- Boolean variables—this enables the logical reasoning, and
- linear inequalities as constraints—this enables the arithmetic reasoning.

Hence, for example, typical PB constraints include "cardinality constraints":

$$\sum_{i=1}^{N} l_i \geq K \tag{4.1}$$

where the literals l_i are all 0 or 1.[1]

Note that such a constraint can be converted to a set of SAT clauses, but we can end up with an exponential number of clauses. Hence, PB is exponentially more concise than SAT. This is a significant advantage in itself. (Actually, if we allow the introduction of new variables, then we can convert it to a polynomial number of clauses, but in practice the extra variables confuse solvers, leading to poor performance.) We also allow "full-PB" constraints:

$$\sum_{i=1}^{N} w_i l_i \geq K \tag{4.2}$$

in which we are allowed a weight w_i for each literal in the sum. This is particularly useful when encoding optimization problems—e.g., the weights can correspond to costs of tasks associated with the literals.

In the OR world, this representation is already well-known as "0/1 programming." We use the term pseudo-Boolean to emphasize that our approach to solving problems is based on Boolean solvers (SAT solvers) rather than standard OR methods.

OR solvers are usually based on solving "linear relaxations" of problems; that is, the requirement that variables be integer-valued is dropped, giving a linear programming (LP) problem, for which polytime solvers are available. However, such solvers are still fairly slow, and so the overall emphasis is that of a search with substantial reasoning at each node, and so the expectation that the nodes studied per second will be small. By comparison, the methods underlying SAT solvers (to be discussed later) tend to involve very lightweight reasoning; the reasoning captures less information, but this is potentially compensated by a much higher search speed—more nodes per second. Also, since the OR methods are so reliant on the LP relaxation it is essential that the encoding be "LP- tight", that is, the LP relaxation should be fairly informative. Unfortunately, for SAT problems, the LP relaxation is useless: this is reflected in the fact that SAT solvers are needed at all—SAT is a subset of IP, and if OR solvers performed well on this subset then specialized SAT solvers would be redundant. Hence, for problems with a large number of logical constraints, we should expect that LP-methods will be ineffective, and that logical methods will be needed. Even for standard OR problems it can take a significant effort by a domain/OR expert to find an encoding that is LP-tight, making automatic production of useful encodings much harder. One of our hopes is to reduce the reliance on LP-tight encodings by using logic-based solvers, and so make it easier to produce automatic encodings.

In relatively unstructured problems (as from a heterogeneous environment, with few clean rules), it may well be easier to live with a PB encoding rather than expend significant manual effort looking for a clever (but fragile) IP encoding.

Hence, our overall aim has been to take AI search methods developed for SAT and adapt them to the better PB representation.

[1] A literal is either a variable x or its complement $\bar{x}$, defined to be $1 - x$.

The resulting solver for decision problems is called PARIS (for some relevant background see [7]). We also give a description of oPARIS, a version of PARIS designed to handle optimization problems, which also incorporates significantly better heuristics.

The specific context of use is to take problems from the ATTEND encoder that represent (a subset of) a SNAP problem. [3]

4.2 Basic PARIS

PARIS is basically a direct reimplementation (by Heidi Dixon) of zCHAFF ([15] the best SAT solver at the time) but with the addition of support for cardinality and full-PB clauses. The paper [7] actually discusses an obsolete version of PARIS that was based on RELSAT [2] rather than CHAFF, however the high-level ideas are the same).

The two most important features to get right in a SAT-based solver are fast propagation and the ability to learn constraints when backtracking. We discuss each in turn.

4.2.1 Propagation

Unit propagation is the process of finding all cases in which the combination of a clause and the current partial assignment P force a new literal; for example if we have the clause $x \vee y$ and $\bar{x} \in P$ then we can deduce y and extend P.

It is critical that unit propagation be efficient. Clearly, it is possible to propagate by simply walking every clause, however, there are better methods (now common in SAT solvers):

- **counting based methods:** for each clause, maintain a count of the number of valued, and the number of true, literals. A clause can only lead to propagation (or contradiction) if it has zero true literals and at most one unvalued literal. Hence there is no need to walk the clause until the counting tests are met. This is better than constantly having to walk the clause, but the cost of maintaining the counts is still high.
- **watched literal methods:** developed for SAT solvers [27] and now the generally preferred approach, this relies on the observation that if two of the literals in a clause are unvalued then the clause cannot cause propagation, independently of the value of any of the other literals. Hence for every clause just two literals of the clause are marked to be watched. The clause is not inspected until the search inspects one of these two literals—in particular, changes to other literals in the clause do not require any action. This saves many cycles. If the search does cause one of the watched literals to be changed, then the clause might have to be walked, and new literals selected for watching, but in practice, this method is still a significant win.

It is straightforward to extend the watched literals method to handle cardinality constraints. For a constraint of the form:

$$\sum_{i=1}^{N} l_i \geq K \tag{4.3}$$

propagation cannot happen if (K+1) or more literals are unvalued, and so we simply watch K+1 literals within the constraint.

However, for full PB constraints we need to take account of all the weights of the literals and so we use a counting-based method. It is useful to talk about the deficit of a clause as being the value of the LHS minus the RHS, that is $\sum_i w_i - K$ for equation 4.2. Then we maintain:

- **poss**: the highest potential value of the deficit. Note that $poss \geq 0$ is necessary for the clause to be satisfiable. The value of $poss$ can be used to detect when propagations will happen, or when the clause forces a backtrack.
- **curr**: the lowest possible value of the deficit. $curr$ can only increase as literals are valued. If $curr \geq 0$ then the clause is satisfied.

Indexing structures are maintained so that we can quickly find all the clauses that contain a literal and adjust the counts.

4.2.2 Clause Learning

Learning during the reasoning process can make it possible for the solver to avoid repeating unproductive work. The basic idea is to take the clauses involved in a contradiction and combine them to produce a new clause. In particular, the search will discover a backtrack when it finds a constraint that forces some variable to have a value opposite to that already assigned to it. The variable is called the conflict variable, the clauses that force it to have opposite values are called conflict clauses. In SAT, the two conflict clauses are resolved together to produce a new learned clause, and the backtrack corresponds to performing a sequence of such resolutions. (Note that while most SAT solvers do not explicitly do resolution, they do other reasoning that has the same net effect as a set of resolutions.)

The basic idea in PARIS is to make the inference within the constraint learning become explicit, and to replace the inference by resolution with inference by a reasoning system appropriate to PB, specifically that of "cutting planes". Cutting-plane reasoning has two simple components:

- taking linear combinations of inequalities:
  e.g., from $2x + y \geq 2$ and $x + y \geq 1$, derive $3x + 2y \geq 3$
- rounding: e.g., from $2x + 2y \geq 3$ we have $x + y \geq 3/2$, but because variables are 0/1 we must also have $x + y \geq 2$.

However, there are still some choices about what should be learned, and the user can select among these. Options are:

- **learn CNF**: learn a simple clause. The primary purpose of this is to measure whether or not learning of non-CNF is advantageous. However, it is possible that on some instances this might be useful, simply because it might reduce some of the overhead of learning.
- **learn cardinality:** learn the best cardinality constraint. If the conflict clauses, or the learned clause, is full PB then some clauses are weakened until they become cardinality constraints (the weakening is done in such a fashion as to preserve the learning needed to drive the backtracking). This learning method has the disadvantage that the weakening may discard some chance to learn something important. However, it has the advantage that the coefficients are always one, and so the constraints remain relatively simple.

- **learn PB:** allow the inference to produce a full PB formula. This is not always possible, as sometimes the learned clause might be satisfied.[2] The reason for this is that sometimes the the conflict variable actually has a consistent real value (i.e., 1/2); since the learned clause does not contain the conflict variable, it cannot care whether the value is real or Boolean; and so the learned clause must be satisfiable. This argument was discovered independently by Chai and Kuehlmann [4].
- **learn "best"** infer multiple clauses (such as both cardinality and full-PB), and pick the one that gives the largest backjump.

Initial expectations were that full PB or "best" learning would be the best performing, but, somewhat surprisingly, cardinality learning generally out-performs them. It is not yet understood why, and might be just a side-effect of other heuristics. Certainly, there is significant potential for improvement.

Note that the SAT solvers are based on the DPLL [6, 5] algorithm, and that this algorithm was extended to PB by Barth, in the solver "opbdp" [1]. Both oPARIS and opbdp handle the same input problems. The essential difference between them is that oPARIS incorporates clause learning; without this the latest methods of SAT solvers cannot be used, but more importantly, it is the learning of PB clauses and not just CNF (or no learning at all) that can really take advantage of the PB representation. For example, even though opbdp works with the PB representation it will still take exponential time on the PHP. In contrast, PARIS takes polynomial time.

4.3 Optimization with PARIS: oPARIS

The point of oPARIS is to extend PARIS to handle optimization problems where, in addition to the constraints, we are given an objective function, Q, a linear sum of literals representing the quality of the solutions:

$$Q = \sum_i w_i l_i \tag{4.4}$$

The intention is to maximize Q. That is, to solve problems expressed in the form:

$$\begin{array}{rl} & max \quad Q \\ \text{such that} & \\ & \Gamma \end{array} \tag{4.5}$$

where Γ is a set of PB constraints.

Note that if we pick a target quality, q, then demanding that it is achieved is simply equivalent to adding the constraint $Q \geq q$, which is itself a PB constraint, so we can solve $\Gamma \wedge (Q \geq q)$ using PARIS.

Hence oPARIS works by simply using PARIS in order to solve a sequence of problems in which solution quality is forced to increase.

Procedure 4.3.1 (oPARIS) *Given a PB maximization problem, compute a solution with maximal quality:*

[2] As first observed by Heidi Dixon.

```
1   q ← 0
2   Γ_W ← Γ                                          # Constraint Set
3   M_B ← none found                                 # Best Model
4   while (1)
5         Γ_W ←  Γ_W ∧ (Q ≥ q)
6         M_W ← PARIS(Γ_W)
7         if ( M_W is failure )
8           then
9                 return M_B
10          else                                     # M_W is the new best
11                M_B ← M_W
12                q ← quality(M_W)
13                if ( q is already maximal ) return M_B
14                q ← q + 1
```

The constraint is tightened on line 5, unless (on line 13) we already know that the objective is maximally satisfied (all the literals in Q were already true), and no improvement is possible.

Note that the calls to PARIS have the side effect of adding new derived clauses to Γ_W during the search. However, the logic is monotonic—adding a constraint never invalidates a derived clause. Hence, trivially, there is no need to delete learned clauses when we tighten. This means that some of the work used to derive early solutions can be used to prune the search on subsequent calls to PARIS.

In practice, the call to PARIS is given a time limit; if the limit is reached then the returned M_B is reported as the best found, rather than the optimal.

4.4 Heuristics Exploited

In moving from PARIS to OPARIS we also extended, and improved the implementation of, many of the standard heuristics. These include the familiar SAT technique:

- **rapid restarts:** the search is repeatedly restarted at the root node. It has long been known that restarts can significantly improve the performance of depth-first search [12]. More recently, restarts have very effectively been combined into algorithms with learning [11].

and the usual methods for selecting branch variables:

- **VSIDS:** from ZCHAFF [15]
- **"Current-Top-Clause" (CTC):** from Berkmin [10]

Both VSIDS and CTC try to keep the search focused within the region defined by previously learned clauses, rather than jumping around the search space too much. This seems to enhance the effectiveness of the learning.

Somewhat more unusual are methods to keep a history of good values, and methods to efficiently handle pure literals, described below.

4.4.1 History

A well-known technique in the search community is that of keeping a "history", that is, storing the previous choices within the search. This has been particularly well exploited for SAT by the algorithm "UnitWalk" [13]. The "history" value of a variable is stored and is only changed when propagation forces it to the opposite value.

In particular, if a variable is unset during search, for example by a restart, then the history value can be reused. It is reasonable (though unconfirmed) to expect that this functions by:

- allowing the storage of "goods"—if a lot of search effort has gone into finding a solution to sub-problem, then it would be retained, unless forced to change.
- keeping the search focused, by giving some impetus to return to previous regions, and so retaining the relevance of the learned information.

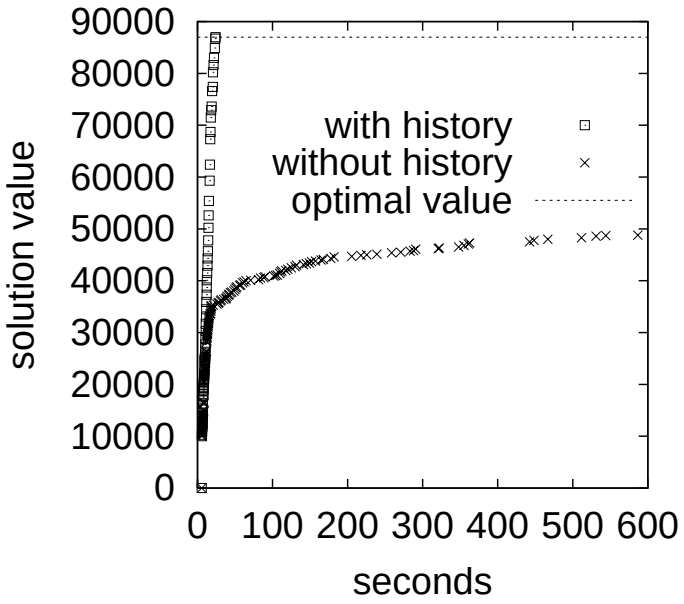

Figure 4.1: Performance with/without the History mechanism

Since both SAT and PB have Boolean variables, it was straightforward to implement this within OPARIS. However, despite the simplicity, it made a large difference. For example, Figure 4.1 shows performance on an instance from ISI, and shows that without the history the search can easily get stuck at a very sub-optimal solution quality.

4.4.2 Structure-Dependent Heuristics

This subsection describes some heuristics that were motivated by the particular structure of ISI instances. In particular, there are variables s_i that are true iff task i is to be performed. Thus many critical constraints have the form:

$$s_i \quad \rightarrow \quad \text{constraint} \tag{4.6}$$

The objective function is of the form:

$$\sum_i v_i s_i \tag{4.7}$$

where v_i is the value of task i.

Pure Literals

A pure literal is a literal that occurs only positively in the (unsatisfied) constraints. In this case, it cannot affect satisfiability to simply enforce the pure literal. It is rare that a literal is pure in the input theory, but literals can become pure at nodes within the search process (all clauses with the bad sign being satisfied because of some other literal), and could be enforced at those node (and all their children). This is a well-known technique in SAT, but has usually been omitted from solvers, because the complexity of testing for pure literals outweighs the advantages from enforcing them.

However, in the ISI instances, turning off a switch means deciding not to perform a task, and so all the variables corresponding to the decision of how to perform that task become irrelevant. However, in the encodings, these irrelevant variables do not disappear, instead they become purely negative, and so the desirable thing to do is to simply set them all to false. Unfortunately, if this is not done, there is a danger that the branch heuristics might consider them for branching. In some cases, the branch heuristics might carefully decide, one-by-one, which of the thousands of pure literals to branch on first. If this is allowed to happen, then even the trivial solution, in which no tasks are enabled, and all constraints ought to be trivially satisfied, can take many seconds to compute. To stop this happening we enable a limited form of pure literal reasoning.

In a preprocessing stage we build "pure literal propagation" rules of the form:

$$x \quad \Rightarrow \quad y \tag{4.8}$$

with the meaning that, if x is enforced then y becomes a pure literal, and so we can enforce y. It is important to note that this is a rule, and not a logical implication—it should not be reversed or used within the learning.

The preprocessing will typically only be done with x taken from a small set. For the ISI instances it is generally only worthwhile to look for rules following from a switch being turned off, that is:

$$\neg s \quad \Rightarrow \quad y \tag{4.9}$$

These rules can be used:

- to prevent a pure literal from being selected for branching,
- to enforce pure literal rules when no other branch variables are available.

These are often essential to save the heuristics from having to make many pointless decisions.

Branching on switch variables

Given the important role played by the switch variable, it can be advantageous to encourage the search to branch on them. Hence, methods are supplied to give an extra weight to selected groups of variables.

4.5 Results

Firstly, we remark that it makes little sense to compare with SAT solvers because of the difficulty of converting to sensibly-sized SAT representation. In the ISI instances, the weights in the objective function are quite large and so make a SAT equivalent very large and unwieldy.

Instead, the main practical alternative to OPARIS is the local search algorithm WSAT(OIP) [26], which works with the same representation and does iterative repair. Although it has the disadvantage of being an incomplete solver (it can never prove a solution is optimal) it can often be highly effective.

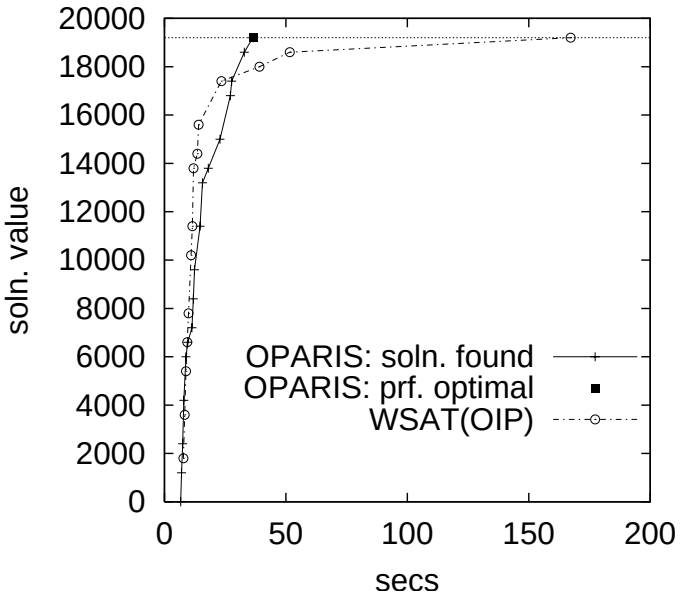

Figure 4.2: OPARIS vs. WSAT without the pilot qualifications extension.

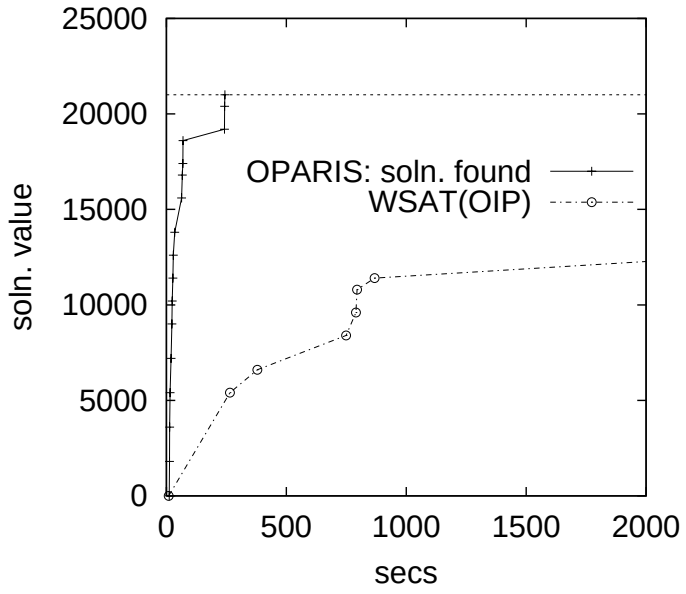

Figure 4.3: OPARIS vs. WSAT(OIP) with the pilot qualifications extension.

The relative performance of OPARIS and WSAT(OIP) on instances from ATTEND depends a lot on whether or not the instances include the addition of extensions needed to handle pilot qualifications [3]. For example, Figure 4.2 gives performance on an instance without the extensions. The optimal value is 19200 and OPARIS both finds the optimal and proves it optimal before

WSAT(OIP) even finds it. However, in this case, in the earlier stages of the search the performance of WSAT(OIP) is better than OPARIS. This seems to be fairly typical behavior—in the easy early stages nonsystematic search is faster, but as the problem gets harder the systematic search does better.

Figure 4.3 shows performance on an instance with the extensions. The extensions allow pilot qualifications to be achieved, and now it turns out all tasks can be scheduled, leading to an optimal value of 21000. In this case, WSAT(OIP) does not even do well during the early stages. We think this is because the pilot qualifications extension corresponds roughly to a planning problem, with long chains of reasoning, and it is known that WSAT does poorly with long implication chains. In contrast, the propagation-based search of OPARIS handles long implication chains naturally.

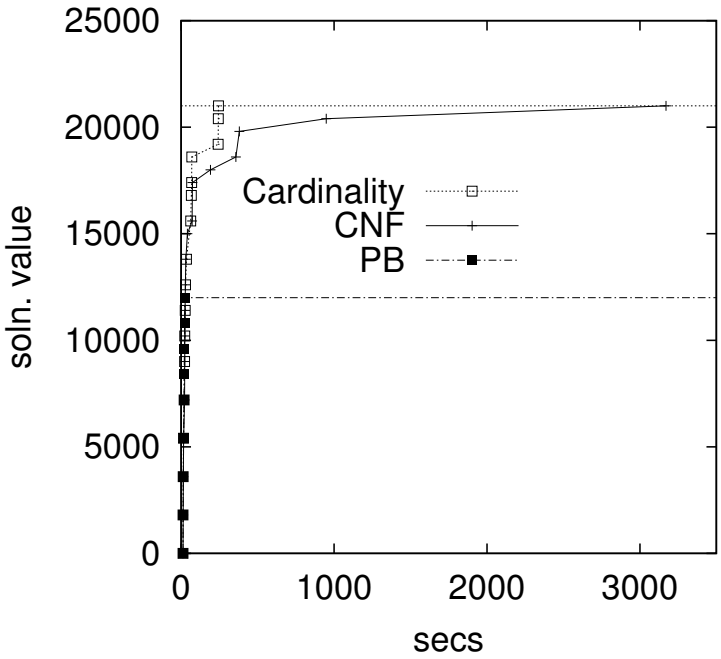

Figure 4.4: Example of effects, in OPARIS, of learning simple CNF clauses versus learning of cardinality constraints. Using Berkmin heuristic.

Finally, figures 4.4 and 4.5 show performance on an instance with CNF learning, PB learning, and cardinality learning. Initial progress does not seem to care about the learning, but when we reach the harder regions learning begins to make significant differences. In particular, in figure 4.4 cardinality learning provides a dramatic improvement. These experiments show, however, that the heuristics used interact strongly with the type of learning chosen in ways that we do not yet fully understand.

4.6 Discussion and Conclusions

SAT solvers perform well on problems that do not require a significant amount of arithmetic reasoning, but can lose out exponentially badly on problems, such as PHP, in which even the simplest arithmetic can be of help. To remedy this we have taken the underlying methods of SAT solvers,

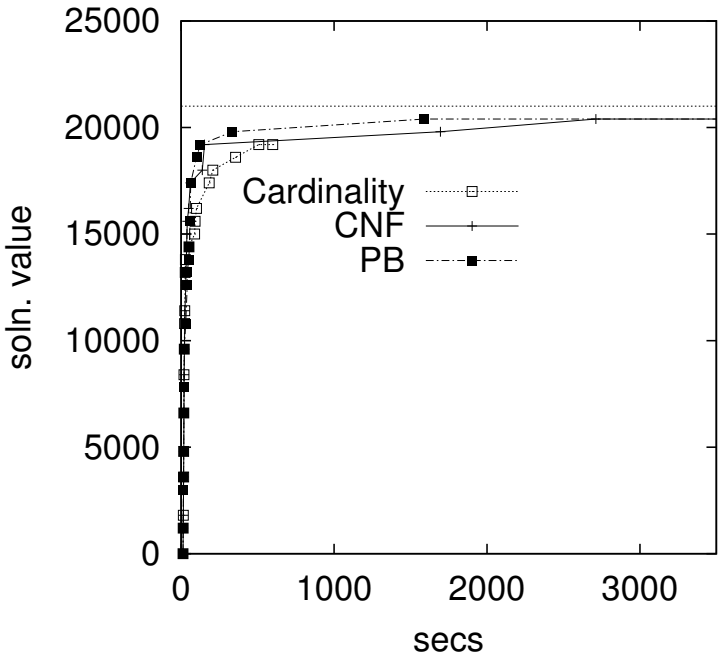

Figure 4.5: Example of effects, in OPARIS, of learning simple CNF clauses versus learning of cardinality constraints. Using VSIDS heuristic.

and modified them for the pseudo-Boolean representation: In particular, clause learning is modified to allow the learning of PB clauses and so gain the power of some arithmetic reasoning.

There are many possible avenues for future work, but of these two stand out: making the search algorithm adaptive to the time bound, and "lifting" the representation so as to make it more concise.

4.6.1 Using the time bound

Usually, on realistic problems, OPARIS is not going to be able to prove optimality, and so must be given an explicit time bound (returning the best solution found on timing out). It would be standard for such an anytime algorithm with a time bound (a contract algorithm in this case) to reason based on the time bound and so select or adjust the heuristics so as to find better solutions. This is currently not done for two reasons. Firstly, the heuristics are relatively new and, though effective, they are poorly understood. This makes it hard to obtain the necessary predictions of their effects on "performance profiles". Secondly, some preliminary experiments suggested that any exploitable effects were small; it tended to be that if a heuristic worked well, then it worked well over all time bounds, and so should just be selected uniformly. However, further work on this topic is warranted.

4.6.2 Lifting

Even though the PB representation is significantly more concise than SAT, problem representations can still get quite large, and exhaust available memory. A significant part of the reason for this is that many constraints are most naturally expressed using a "lifted" form. A lifted representation

uses universal quantifiers over domains, e.g.:

$$\forall i, j, k. \quad p(i, j) \vee \neg r(j, k) \tag{4.10}$$

to express natural structural regularities of the domain. Before these representations can be fed to existing solvers such as OPARIS or WSAT(OIP), however, the quantifiers must be expanded, or "grounded out."

This problem is already familiar from work in SAT, and a natural solution is to modify the solvers to work directly with the lifted form. Originally it was believed that this would be too inefficient, however, work at CIRL on QPROP ([19, 9] and surveyed in [8]) showed that use of the lifted form can be efficient. In fact, surprisingly, there are theoretical reasons why it might even be more efficient than using the ground form.

A reasonable goal would be a language that mixes QPROP and PB, "Quantified Pseudo-Boolean", QPB, and so for example allows constraints such as:

$$\forall i, j, k. \quad 2p(i, j) + r(j, k) + q(r, k) \geq 2 \tag{4.11}$$

This is especially reasonable as it is essentially (a subset of) the language of OR modeling frameworks such as AMPL,[3] and so there is already evidence as to the utility of the representation.

[3] http://www.ampl.com

Bibliography

[1] P. Barth. A Davis-Putnam based enumeration algorithm for linear pseudo-boolean optimization. Technical Report MPI-I-95-2-003, Max Planck Institute, 1995.

[2] R. J. Bayardo and R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI–97)*, pages 203–208, Providence, RI, 1997.

[3] A. Bugacov, D. Kim, R. Neches, and G. Pike. ATTEND: Analytical Tools To Evaluate Negotiation Difficulty, 2003. http://www.isi.edu/attend/

[4] D. Chai and A. Kuehlmann. A fast pseudo-boolean constraint solver.

[5] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Journal of the ACM*, 5:394–397, 1962.

[6] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[7] H. E. Dixon and M. L. Ginsberg. Inference methods for a pseudo-boolean satisfiability solver. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI–2002)*, pages 635–640, 2002. http://www.cirl.uoregon.edu/dixon/

[8] H. E. Dixon, M. L. Ginsberg, and A. J. Parkes. Likely near-term advances in SAT solvers. In *Workshop on Microprocessor Test and Verification (MTV'02)*, June 2002. http://citeseer.nj.nec.com/dixon02likely.html

[9] M. L. Ginsberg and A. J. Parkes. Satisfiability algorithms and finite quantification. In A. Cohn, F. Giunchiglia, and B. Selman, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Seventh International Conference (KR2000)*, pages 290–701. Morgan Kaufmann, 2000.

[10] E. Goldberg and Y. Novikov. Berkmin: A fast and robust SAT solver. In *Design Automation and Test in Europe (DATE)*, pages 142–149, 2002.

[11] C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98)*, pages 431–437, Madison, Wisconsin, 1998. http://citeseer.nj.nec.com/gomes98boosting.html

[12] W. D. Harvey. *Nonsystematic Backtracking Search*. PhD thesis, Stanford University, 1995.

[13] E. Hirsch and A. Kojevnikov. UnitWalk: A new SAT solver that uses local search guided by unit clause elimination. PDMI preprint 9/2001, Steklov Institute of Mathematics at St.Petersburg, 2001.

[14] D. Mammen and V. Lesser. Problem structure and subproblem sharing in multi-agent systems. In *International Conference on Multi Agent Systems (ICMAS 98)*, 1998. http://citeseer.nj.nec.com/mammen98problem.html

[15] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *39th Design Automation Conference*, 2001.

[16] R. Neches and P. Szekely. CAMERA: Coordination And Management Environments for Responsive Agents. http://www.isi.edu/camera/

[17] C. Papadimitriou. On selecting a satisfying truth assignment. In *Proc. IEEE symposium on Foundations of Computer Science*, pages 163–169, 1991.

[18] A. J. Parkes. Clustering at the Phase Transition. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 340–345, Providence, RI, 1997. http://www.cirl.uoregon.edu/parkes/

[19] A. J. Parkes. *Lifted Search Engines for Satisfiability*. PhD thesis, University of Oregon, June 1999. Available from http://www.cirl.uoregon.edu/parkes

[20] A. J. Parkes. Distributed local search, phase transitions, and polylog time. In *Proceedings of the workshop on "Stochastic Search Algorithms", held at "Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01)"*, August 2001. http://www.cirl.uoregon.edu/parkes/

[21] A. J. Parkes. Exploiting solution clusters for coarse-grained distributed search. In *Proceedings of the workshop on "Distributed Constraint Reasoning", held at "Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01)"*, August 2001. http://www.cirl.uoregon.edu/parkes/

[22] A. J. Parkes. Easy predictions for the easy-hard-easy transition. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-2002)*, July 2002. http://www.cirl.uoregon.edu/parkes/

[23] A. J. Parkes. Scaling properties of pure random walk on random 3-SAT. In *Proceedings of Eighth International Conference on Principles and Practice of Constraint Programming (CP 2002)*, volume LNCS 2470 of *Lecture Notes in Computer Science*, pages 708–713, 2002. http://www.cirl.uoregon.edu/parkes/

[24] C. van Buskirk, B. Dawant, G. Karsai, J. Sprinkle, G. Szokoli, K. Suwanmongkol, and R. Currer. Computer-aided aircraft maintenance scheduling. Technical Report ISIS-02-303, Vanderbilt University, November 2002.

[25] C. van Buskirk, B. Dawant, G. Karsai, J. Sprinkle, G. Szokoli, K. Suwanmongkol, and R. Currer. MAPLANT: Maintenance Planning Tools, 2003. http://www.isis.vanderbilt.edu/maplant/doc/build/html/MAPLANT/about.html

[26] J. P. Walser. Solving linear Pseudo-Boolean constraint problems with local search. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI–97)*, pages 269–274, 1997.

[27] H. Zhang and M. E. Stickel. Implementing the Davis-Putnam method. *Journal of Automated Reasoning*, 24(1/2):277–296, 2000.